Teleportation as a Strategy for Improving Concurrent Skiplist Performance

by

Frances Steen

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science

at

BROWN UNIVERSITY

May 2015

Author .....

Frances Steen
Department of Computer Science
April 15, 2015

Advised by .....

Maurice Herlihy
Professor of Computer Science
Thesis Supervisor

Read by .....

Thomas Doeppner
Vice Chairman & Director of Undergraduate Studies
Department of Computer Science

Teleportation as a Strategy for Improving Concurrent Skiplist Performance

by

Frances Steen

Submitted to the Department of Computer Science
on April 15, 2015, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science

Abstract

We explore how the use of teleportation can contribute to better skiplist performance. Teleportation is a technique that leverages transactional memory to minimize the costs incurred through intermediate node traversals in concurrent linked-node data-structures such as linked-lists and skiplists. Intel's development of the Haswell microarchitecture, which has hardware support for transactional memory, makes teleportation a more viable strategy for widespread use. We apply the strategy of teleportation to lock-coupled skiplists and hazard-protected lazy skiplists, and measure the changes in performance that result. We find that teleportation becomes costly for the lock-coupled skiplist, as it is used in high-contention environments where lock acquisition is required within transactions, but contributes to performance gains for the hazard-protected lazy skiplist, where lock acquisition does not occur within transactions. Our results provide new insights into the types of data-structures that are good candidates for performance improvements through the use of teleportation.

Thesis Supervisor: Maurice Herlihy
Title: Professor of Computer Science

Acknowledgments

Thanks to my thesis advisor, Professor Herlihy, whose insights and answers to my questions were essential in moving my thesis along. Thanks, also, to my parents Kyra and Paul, and to my friends, Sonia and Jared, for proofreading my thesis and for letting me bounce ideas off of them throughout the year.

Contents

Introduction

The focus of this thesis is on improving the performance of the concurrent skiplist. As transactional memory becomes increasingly widely available, we can begin to design general-purpose concurrent data-structures and algorithms that leverage this relatively new concurrency control mechanism. This thesis is motivated by previous work on concurrent linked-lists [2], in which teleportation is used to produce speed-ups. In this chapter, we give background on transactional memory, the skiplist data structure, and two specific types of concurrent skiplist: the lock-coupled and the lazy skiplist. When we discuss the lock-coupled and the lazy skiplist, we also give an overview of what teleportation is and how we expect it to speed up these two skiplist types.

1.1 Transactional Memory

Much like locks and synchronization primitives, transactional memory is used as a tool for enabling thread-safe concurrent programming. Transactions are the means through which transactional memory concurrency control is implemented. Whereas locks enable thread-safe concurrent programing pessimistically, by preventing concurrent access to shared memory locations under the assumption that unprotected shared memory locations will be accessed in an unsafe manner, transactional memory operates under an optimistic assumption: assume shared memory is being accessed in a safe manner until something bad happens, and then deal with the consequences. More precisely, transactional memory allows a block of multiple instructions inside of a transaction to be executed atomically. This is done by executing the code within the transaction block speculatively. If a conflict occurs, then the changes that were made during the transaction are rolled back. Otherwise, in the case of no conflict, the transaction is committed and the writes to shared memory locations within the transaction appear to take place atomically.

In order to track whether a conflict occurs, the locations that are read within

the transaction block are added to the transaction's read-set and the locations that are modified within the transaction block are added to the transaction's write-set. If a different thread modifies a location tracked within the read-set or the write-set while the transaction is ongoing, then a conflict occurs and the transaction aborts. An abort also occurs if a different thread reads a location that is tracked within the write-set. In addition to conflict aborts, a transaction may abort for several other reasons. For our purposes, the other relevant abort types are: capacity aborts, which occur when the number of memory locations accessed within the transaction exceeds the transaction's capacity; unknown aborts; and user-defined aborts, which are triggered by the programmer within the code. When a transaction aborts, it writes the relevant abort code to its status flag. After an abort occurs, the transaction may be retried or execution may move on to an alternate piece of non-transactional fallback code. Typically, code is written such that transactions are retried several times but eventually fall back to alternate, non-transactional code following a certain number of aborts. It is necessary that every transaction has a equivalent non-transactional fallback because transactions are not guaranteed to ever commit.

Our code is written using Intel's Restricted Transactional Memory (RTM) interface, available on its Haswell microarchitecture. The interface exposes four methods: `xbegin`, which begins a transaction; `xend`, which ends a transaction; `xabort`, which the programmer can use to abort a transaction (the programmer also supplies it with an abort code to be written to the status flag); and `xtest`, which reports whether or not it is being called from within a transaction.

1.2 The Skiplist

The skiplist is linked-node data-structure that is made up of several linked-lists. It shares the same functionality as the linked-list. It stores a set of unique elements in increasing order and has add, remove, and contains methods. The first two methods mutate the set and the third checks for set membership. All three methods have expected logarithmic run times – a substantial improvement in performance over the linear run time methods of the linked-list. Additionally, unlike other data-structures with logarithmic time search methods (such as trees), the skiplist does not require rebalancing, which makes it easily adaptable to concurrent environments.

The logarithmic performance of the skiplist is a direct result of its structure. It is made up of several levels of linked-list, with the bottom-level linked-list containing all of the elements in the set. As the level of the skiplist increases, the linked-lists

grow sparser, because each linked-list skips over some of the nodes contained in the linked-list at the level below it. Thus, a linked-list at a given level is a subset of all of the lower-level linked lists. We determine which nodes are contained in the linked-list at a given level based on node height. Upon node creation, a node's height is chosen from an exponential distribution, where the probability that the node will appear at level i is p^i. Thus p, $0 < p < 1$, is the expected fraction of nodes from a given level that will be skipped over at the next higher level. (The probability of a node appearing at level 0 – the bottom level – is 1.) When p is $\frac{1}{2}$, for example, each linked-list in the skiplist contains half as many nodes as the level below it. This gives the skiplist an expected logarithmic depth. Additionally, the skiplist has head and tail sentinel nodes which have the maximum heights and which bookend the other nodes in the skiplist.

For our purposes, we use an underlying find method to search the skiplist in the add, remove, and contains methods. The find method locates the nodes that come just before the location of the value for which we are searching in the skiplist. We call these nodes the predecessor nodes, and we call the node whose value matches that for which we are searching the victim node. To find the victim node, traversal begins from the head node and the top level of the skiplist. This level of the skiplist has the fewest nodes linked into it. We traverse across this level until we locate the victim's predecessor node at this level of the skiplist, whereupon we move down a level in the skiplist and continue our traversal. We repeat this process until we reach the victim's predecessor node at the lowest level in the skiplist. We expect to traverse a constant number of nodes at each level of the skiplist, and we expect our skiplist to have depth that is logarithmic in the length of the skiplist. Thus, our find method has an expected logarithmic run time.

The add and remove methods rely heavily on the find method. In both cases, we use the find method to locate the nodes that point to the spot where we will either add a node by linking it in or remove a node by redirecting the next pointers of the predecessors that we found to point to the node after the one we are removing. More information about the skiplist can be found in [1, ch.14].

1.3 The Lock-Coupled Skiplist

The lock-coupled skiplist is the first of two versions of thread-safe skiplist that we look at. In the most basic lock-coupled skiplist, each node in the skiplist has its own lock. To ensure correctness and prevent deadlock during concurrent add, remove and

contains calls, nodes are traversed in a hand-over-hand manner. That is, when nodes are traversed across a single level, the lock for a successor node is obtained prior to the release of the lock for the current node. This ensures that, for a traversal of the skiplist by a given thread, at least one node remains locked by the thread at all times.

While hand-over-hand locking is necessary to ensure correctness during concurrent traversals and mutations of the skiplist, it contributes significant overhead to skiplist traversals. Each thread must obtain a lock for every node that it traverses and threads traversing the skiplist encounter a significant sequential bottleneck because it is impossible for one thread to pass another thread during traversal. Thus, a thread searching for a high-valued node must inevitably wait behind a thread performing an add or delete on a low-valued node before its search can continue. Similarly, if the scheduler pauses a thread that happens to hold the lock to the first node in the skiplist, all other threads are prevented from traversing the skiplist.

The key observation for this research is that it seems possible to alleviate this problem using transactions. When node traversal occurs within a transaction, it becomes unnecessary to obtain a lock for every node that is traversed. Instead, it suffices to "teleport" the locks across several nodes – to traverse the nodes as usual, but without obtaining any locks. At the end of the transaction, the node that was initially locked is unlocked and the final node to be traversed is locked. (Transactions let us eliminate intermediate lock acquisitions because everything that occurs within the transaction appears to take place atomically, so it is not possible for other threads to interfere with our node traversal. In actuality, as we discussed in the Transactional Memory section, if there is an interference with a memory location that is tracked in the read-set or write-set of the transaction, then the transaction will abort.) Lastly, prior to committing, we lock all predecessors to the victim node that were encountered during the teleport.

Using the lock teleportation strategy, we reduce the number of lock acquisitions required in a skiplist traversal and we enable threads to pass one another during traversal. (Passing is possible because a thread searching the skiplist within a transaction can traverse locked intermediate nodes.) However, as we will see later, further changes are required in order to make lock teleportation a viable strategy for improving the performance of lock-coupled skiplists.

1.4 The Lazy Skiplist

The lazy skiplist is the second type of skiplist that we explore. It is largely the same as the lock-coupled skiplist in structure (each node has its own lock), but differs from the lock-coupled skiplist in locking algorithm. Instead of using hand-over-hand locking, in the lazy skiplist's find method we traverse nodes without obtaining any locks. When we need to add a node, we lock all of its predecessors and then validate that they have not changed before linking the node into the skiplist. (This is necessary because other threads are mutating the skiplist concurrently and may have added or deleted nodes such that the set of predecessors is affected.) Similarly, when we need to remove a node, we ensure that it is still logically a part of the skiplist before locking it and then, as with the add method, validating that its predecessors have not changed. We then logically remove the node from the skiplist, by setting a flag that indicates that it has been deleted, before physically unlinking it.

The lazy skiplist as described above is already rather efficient. However, we are also concerned with memory-management. In the lock-coupled skiplist, memory-management is not an issue, as deleted nodes can be immediately recycled back into the skiplist. (We note that if the skiplist's contains method is lock-free, then slightly more work is required.) For this reason, we do not implement node-recycling for the lock-coupled skiplist, and maintain our focus on teleportation instead. However, node-recycling is not as trivial for the lazy skiplist. In the lazy skiplist, it is possible for a thread to retain a reference to a node that has already been removed. Thus, care must be taken to ensure that such a node is not recycled back into the skiplist until all threads have discarded their references to it. To do this, we use hazard pointers, which are node references that each thread publishes to a common location, indicating that the node is not safe to recycle. Before reusing a node, a thread must check the hazard pointer array to ensure that the node is not hazard protected. Additionally, in order to ensure that this information is immediately visible to all threads, we must follow each hazard pointer publication with a memory barrier. A thread overwrites its hazard pointers when a new node needs to be hazard protected.

The use of hazard pointers allows us to recycle nodes in the lazy skiplist. However, it creates significant overhead for the lazy skiplist because hazard protection must be performed for each node prior to its being read. This is especially expensive because of the necessity of a memory barrier after each hazard pointer publication. We can alleviate this issue using hazard pointer teleportation in much the same way that lock teleportation is employed in the lock-coupled skiplist. That is, we can avoid publishing

a hazard pointer for every node by traversing nodes within a transaction. Just before committing the transaction, hazard pointers are published for the final two nodes that are traversed. The strategy of using hazard pointer teleportation significantly reduces the number of hazard pointers that need to be published, thereby reducing the number of necessary memory barriers and improving performance. We will see later that the lock-free nature of hazard pointer teleportation makes it a successful means of speeding up the lazy skiplist.

Teleporting Lock-Coupled Skiplists

In this chapter, we present an implementation, written in `C++`, of the lock-coupled skiplist that we adapted to use lock teleportation. First, we cover the design of the skiplist and its nodes, followed by an overview of our teleportation algorithm. We then present several variations on the basic teleporting lock-coupled skiplist. We describe the benchmarks that we used to measure the performances of the various lock-coupled skiplists, followed by the results that we obtained and a discussion of these results. In our discussion of the results, we provide an explanation for our finding that the teleporting lock-coupled skiplist remains slower than its non-teleporting counterpart. Finally, we suggest changes that might produce a better teleporting lock-coupled skiplist and that motivate our design of the teleporting lazy skiplist.

2.1 Design of Skiplist and Nodes

Here, we present an overview of the design of the lock-coupled skiplist and its nodes. In our skiplist implementation, we limit ourselves to dealing with integer values and we maintain a set of nodes whose values are unique. In addition, our skiplist maintains the set in increasing order at all times. The `Skiplist` structure has a reference to a *head* `Node` and a *tail* `Node`. These sentinel `Node`s are given the placeholder values of *MIN_INT* and *MAX_INT*, respectively; they serve to simplify the code. In the `Skiplist` structure, we also specify a *MAX_HEIGHT* for the `Node` structures. The *MAX_HEIGHT* determines how many levels the `Skiplist` can have.

Each `Node` structure consists of several members. An `Int` key stores the value of the `Node`. Each `Node` has an `Int` height, which is determined at the time of `Node` creation by sampling from an exponential distribution across the possible heights. A `Lock` serves to protect the `Node` from concurrent accesses. Lastly, each `Node` has an array of pointers to next `Node`s, one pointer for every level up until the `Node`'s height. Note that a `Node`'s next pointer at a given level must point to a `Node` whose height is at least 1 greater than that level. (This is because levels are zero-indexed and we

cannot have `Node`s that are linked in to the skiplist at levels above their heights.)

2.2 Algorithm

We now turn our attention to the algorithm that we use in the `Find` method, giving specific attention to our procedure for teleportation. For background information on the lock-coupled skiplist, we refer the reader to [1, ch.14].

2.2.1 Find Method and Teleportation

The skiplist has public `Add`, `Remove` and `Contains` methods. These are backed by an underlying `Find` method (see Figure 2-1). The `Find` method takes in: an `Int` value, v; an array of node pointers, *preds*; and a `ThreadLocal` structure, *threadLocal*. It does not return anything. (The `ThreadLocal` structure simply stores information about the thread that is currently executing the code; for our purposes it is only important to note that this structure contains the thread ID.)

In the `Find` method, we search the skiplist for the victim node (the node whose value is equal to v), and populate the *preds* array with references to the (locked) predecessors of the victim. Note that even if the victim node is absent from the skiplist, we still can (and do) find all of its predecessors. (The predecessors to an absent victim are simply those nodes that *would* point to the victim if it were present.) This process encompasses the bulk of the work of mutating the skiplist, leaving the `Add` and `Remove` methods mainly only the work of manipulating pointers to link or unlink a node within the skiplist.

In broad terms, we describe the `Find` method as follows. To begin, we initialize our starting level to the top level of the skiplist. (Recall the skiplist `Find` algorithm traversal pattern described in the previous chapter.) We preliminarily set the top level predecessor to point to the *head* of the skiplist and we lock this node. Inside of a loop, we repeatedly attempt to advance down the skiplist using lock teleportation. If a certain number of attempted transactions abort, we then revert to hand-over-hand locking, which is the fallback traversal strategy. In the most basic implementation, after we advance one node down the skiplist using hand-over-hand locking, we attempt to use lock teleportation again. (However, it is possible to alter the traversal strategy so that, within a `Find` operation, lock teleportation is never revisited after we revert to hand-over-hand locking. We find that this strategy tends to produce better performance.)

When traversing the skiplist, we use a thread's *teleportLimit* to determine when to revert to hand-over-hand locking and how much of the skiplist to traverse within one transaction using lock teleportation. The *teleportLimit* is a variable that sets an upper bound for the number of next pointers that the thread may follow before it must attempt to commit a transaction. It is possible that the thread will traverse fewer than *teleportLimit* nodes; this will be the case when the victim node is found within *teleportLimit* of the starting node. It is important to enforce a traversal limit (rather than allowing the whole `Find` traversal to occur within one transaction) because the likelihood of a successful transaction decreases as the number of nodes traversed within the transaction increases. (The decrease in likelihood is due to the fact that longer teleports are more likely to have read-set or write-set conflicts with other threads and are also more likely to suffer from capacity aborts due to having accessed too many memory locations.)

For each thread executing operations on our skiplist, *teleportLimit* is initialized to some starting value (we use 16 in our code). After every transaction, it is updated using an additive-increase multiplicative-decrease (AIMD) strategy. More specifically, if a transaction utilizes the entire alotted *teleportLimit* and commits, then we increment the *teleportLimit*. If the transaction aborts, then we halve the *teleportLimit*. In the case where the transaction commits before reaching the limit, *teleportLimit* remains the same. This strategy enables us to dynamically search for a good traversal limit.

The `Fallback` component of the `Find` method (see Figure 2-2) retains the traversal strategy of the non-transactional lock-coupled skiplist, except that the `Fallback` method encompasses only one individual next pointer traversal. We read the next node in the skiplist in line 1, and, in line 2, determine whether or not we have arrived at our victim. If we have not, then it is safe to traverse further. We lock the node that we just read and, in lines 4-6, only unlock the previous node if it is not the predecessor to the victim at the level above. Finally, in lines 7-8, we update our predecessor array and return *false*, indicating that our traversal is not complete. If, in line 2, we instead determine that we have reached our victim node, then (in lines 9-14) we move down a level of the skiplist, if possible, and indicate whether or not we have completed our traversal.

The lock teleportation code is somewhat more complex. In `Teleport` (Figure 2-3), we begin by resetting the *teleportLimit*, if necessary, and initializing our local variables. In lines 7-15, we traverse the skiplist. This code is very similar to that used for traversal in the non-transactional lock-coupled skiplist, except for the fact that

```
void find(int v, Node* preds[], ThreadLocal* threadLocal) {
1    bool done, success = false;
2    int start_layer = MaxHeight-1;
3    preds[start_layer] = &LSentinel;
4    preds[start_layer]->lock();
5    while (!done) {
6      do {
7        if ((success = (xbegin(threadLocal)) == _XBEGIN_STARTED)) {
8          done = teleport(v, preds, threadLocal, &start_layer);
9        }
10     } while (!done && (success
                 || (threadLocal->teleportLimit /= 2) > 2));
11     if (done) { break; }
12     done = fallback(v, preds, &start_layer);
13     threadLocal->teleportLimit = 2;
14   }
15 }
```

Figure 2-1: The lock-coupled `Find` method, used in `Add`, `Remove`, and `Contains`.

```
bool fallback(int v, Node* preds[], int* start_layer) {
1    Node* curr = preds[*start_layer]->next[*start_layer];
2    if (curr->value < v) {
3      curr->lock();
4      if (*start_layer == MaxHeight-1
           || preds[*start_layer+1] != preds[*start_layer]) {
5        preds[*start_layer]->unlock();
6      }
7      preds[*start_layer] = curr;
8      return false;
9    } else if (*start_layer > 0) {
10     *start_layer = *start_layer-1;
11     preds[*start_layer] = preds[*start_layer+1];
12     return false;
13   } else { return true; }
14 }
```

Figure 2-2: The lock-coupled `Fallback` method, used in `Find`.

```
bool teleport(int v, Node* preds[],
              ThreadLocal* threadLocal, int* start_layer) {
1    if (threadLocal->teleportLimit < 0) {
2      threadLocal->teleportLimit = DEFAULT_TELEPORT_DISTANCE;
3    }
4    int dist = threadLocal->teleportLimit;
5    int curr_layer;
6    Node* start, pred = preds[*start_layer];
7    for (int layer = *start_layer; layer >= 0 && dist >= 0; layer--) {
8      curr_layer = layer;
9      Node* curr = pred->next[layer];
10     preds[layer] = pred;
11     while (curr->value < v && --dist >= 0) {
12       preds[layer] = pred = curr;
13       curr = curr->next[layer];
14     }
15   }
16   bool done = curr_layer == 0 && preds[0]->next[0]->value >= v;
17   if (*start_layer == MaxHeight-1 || preds[*start_layer+1] != start) {
18     start->unlock();
19   }
20   Node* justLocked = NULL;
21   for (int layer = *start_layer; layer >= curr_layer; layer--) {
22     if (preds[layer] != justLocked) {
23       preds[layer]->lock();
24       justLocked = preds[layer];
25     }
26   }
27   _xend();
28   if (dist <= 0) { threadLocal->teleportLimit++; }
29   *start_layer = curr_layer;
30   return done;
31 }
```

Figure 2-3: The lock-coupled `Teleport` method, used in `Find`.

no locks are obtained and that we have an additional stopping condition, *dist*, which we use to enforce the *teleportLimit*. In line 16, we check to see whether our traversal ended because we reached the *teleportLimit* or because we found our victim node (in which case *done* is true). Just as in the `Fallback` method, we unlock the initial node that was locked, given that it is not the predecessor to the victim at the level above (lines 17-19). Then, in lines 20-26, we go back over all of the predecessors that we found during our traversal and lock them, making sure we never lock the same node twice. (The same node may be a predecessor to the victim at more than one level.) If we attempt to lock a node that was locked prior to the transaction beginning or that becomes locked by another thread while the transaction is ongoing, then an abort occurs. After locking is complete, we attempt to commit the transaction, and, upon success, update our starting level and return whether or not our traversal is complete.

2.2.2 Variations

In this section, we discuss one of the main performance issues that arises for the teleporting lock-coupled skiplist that we introduced in the previous section: contention on top-level nodes. This issue serves to motivate our introduction, later in the section, of several other variations on the teleporting lock-coupled skiplist. However, when we look at the graphs, in the Results section, of the original teleporting lock-coupled skiplist and the variations that we introduce, we determine that these strategies do not significantly relieve the issue. Finally, in the Discussion section of this chapter, we analyze why this is the case.

We turn our attention, first, to our method of locking predecessors in the lock-coupled skiplist. Recall that in our `Find` method, we go through the process of locating and locking all of the predecessors to the victim node, starting with the top-level predecessor and continuing until we reach the lowest-level predecessor. Note that because we require each thread to lock the top-level predecessor of its victim node in order to perform a mutation on the skiplist, it is not possible for two or more threads to concurrently add or remove nodes that have the same top-level predecessor. Thus, in the teleporting skiplist described above, the degree of concurrency during `Add` and `Remove` operations is limited by the number of top-level nodes in the skiplist. Furthermore, because we draw node heights from an exponential distribution, we expect to have a very low ratio of top-level nodes to total number of nodes in the skiplist, especially when the skiplist height is high.

The observation that there is likely to be a great deal of contention on the top-level

predecessors within the skiplist motivates our first variation, which we call the exact (lock-coupled) skiplist. The design of the exact skiplist stems from the observation that, in adding or removing a node within a skiplist, it is sufficient to lock only those nodes that are predecessors at or below the top level of the node in question. The reason that we need to lock predecessor nodes in the first place is in order to to prevent changes to these nodes by other threads while the victim node is being linked into or unlinked from the skiplist. However, because we only link or unlink a node up to its top level, there is no need to lock any predecessors above the top level of the node.

We alter the code for the skiplist presented in previous section to reflect this new strategy. The `Add` and `Remove` methods both speculate the heights of their victim nodes. (For the `Add` method, this is the height of the node that it is trying to link in. The `Remove` method speculates the lowest height possible, which is 1.) Both methods then pass this information along to the `Find` method. During the `Find` traversal, we refrain from locking the predecessor nodes until we either reach the speculated node height or encounter the victim node, whichever occurs first. This strategy applies to both teleporting and non-teleporting exact skiplists and greatly relieves contention on top-level nodes, as most skiplist mutations occur on low-height nodes and thus do not require the locking of a high-height predecessor.

We take the exact skiplist strategy one step further by introducing level-locking skiplists. In this variation, every node in the skiplist has one lock for each of its levels, rather than simply having one lock overall. We thereby further reduce contention on high-height nodes by enabling each of their levels to be locked individually. Thus, for example, if a top level node is the predecessor to a node of height 1, it can be locked at its lowest level and left unlocked at all of its higher levels. This allows it to be locked by another thread as the predecessor, at a higher level, for different node. In non-teleporting contexts, this strategy does not require many changes to the skiplist code that we describe in the previous section. However, the logic for using level-locking in conjunction with teleportation is complex and requires a slightly more conservative locking strategy than the one that we outlined at the beginning of this chapter. These trade-offs yield a teleporting level-locking skiplist that often performs worse than both the teleporting exact skiplist and our original teleporting lock-coupled skiplist.

We compare the performance of these two teleporting lock-coupled skiplist variations and the original teleporting lock-coupled skiplist in the Results section.

2.3 Benchmarking

We now discuss the methods by which we measure the performance of our teleporting lock-coupled skiplists. Our benchmarking is done on Brown's Haswell machine, "Spoonbill," which has an Intel Core i7-4770 CPU. The machine has 4 hyper-threaded cores, which enables it to run 2 threads per core, giving it 8 virtual processors. This fact is relevant because some of our skiplists exhibit a dramatic change in performance trend as we go from 8 to 9 threads.

We benchmark each version of the lock-coupled skiplist by populating it with an initial set of elements and then executing a predetermined number of `Add`, `Remove`, and `Contains` operations on it. We are able to vary several parameters in our benchmarking suite: the skiplist initial size, the benchmark test length, the range from which we sample our node values, the skiplist height, the distribution of operations, and the number of threads performing operations concurrently. Even though many of these factors can impact skiplist performance, we keep most of them fixed in our Results section graphs because they do not significantly impact the *relative* performance between skiplist versions. For all of our benchmarking: we initially populate our skiplists with 5,000 nodes prior to beginning the test; our tests remain fixed at a length of 100,000 operations performed on the skiplist; we draw our node values from between 1 and 10,000; and our skiplist height is 32 throughout. We also only present performance graphs for a fixed distribution of operations. Operation distribution refers to the percent of operations that are read-only `Contains` operations (the rest of the operations consist of an equal balance of `Add` and `Remove` operations – e.g. a 50% read probability describes a test with 50% `Contains` operations, 25 % `Add` operations, and 25% `Remove` operations). This *is* an interesting variable to look at, as teleporting skiplists that do produce performance gains tend to most outperform their non-teleporting counterparts when the read probability is lower. However, in our case, we do not gain any critical information about the relative qualities of the different skiplists from varying the read probability, so we fix it at 50% for our Results section. During our testing, we use between 1 and 12 threads. When more than 1 thread is enlisted to perform operations on the skiplist, we assign each thread an equal number of operations such that our total number of operations is met. (We make the simplifying assumption that each thread will perform the same number of operations in roughly the same amount of time.)

We measure the time that each benchmark test takes to run using the built-in `C` function `gethrtime`, which returns the high-resolution time in nanoseconds (we

convert this to milliseconds for our graphs). Our results for each benchmark test consist of the averages obtained from 5 runs of that test. Each thread used in the test records its start time and finish time, and we take an average of the elapsed times for each threads' execution to get our test run time. We obtain the average commit rate of the threads during a benchmark test by incrementing each threads' "transaction started" counter when a transaction is attempted and incrementing the "transaction committed" counter each time a transaction succeeds. Similarly, we have each thread record how far it teleports during each successful transaction, from which information we obtain the average teleport length for a benchmark test.

Here, we present the results of benchmarking the teleporting lock-coupled skiplist. We compare our original teleporting lock-coupled skiplist ("original"), which we introduced in detail at the beginning of this chapter, to our two variations – the teleporting exact skiplist ("exact") and the teleporting level-locking skiplist ("level"). We benchmark these teleporting skiplists against a non-teleporting exact skiplist ("exact no-teleport"), which we had hoped to outperform. Note that graphs pertaining to teleportation-specific measures do not include the non-teleporting skiplist.

2.4 Results

First, we look at performance, which is the measure that we are most interested in. In Figure 2-4, we graph the run time of each of our skiplists as a function of number of threads. The exact no-teleport skiplist maintains a steady run time between 4 and 10 threads, with decreased performance outside of this range, which is as expected. Our original and exact skiplists share this run time trend but do not perform as well. It is reassuring to note that the exact skiplist does indeed outperform the original skiplist, as we expected would be the case because of its more liberal locking policy. However, whereas the exact no-teleport skiplist maintains a flat performance between 4 and 10 threads, the original and exact skiplists grow slightly worse. This could be an indication that their performance is negatively impacted by increased contention with an increase in the number of threads executing concurrently. (We will see that this is, indeed, the case by looking at the next two graphs.) The level skiplist performs rather poorly even with just 1 thread, which tells us that its algorithm is slower than that of the other skiplists. It does, however, experience a fairly steep improvement in performance as we go from 1 thread up to 8 threads, so it seems to be the case that its code is more parallelizable than that of the other skiplists. Nonetheless, as soon as the number of threads moves beyond the number of virtual processors, we see a

sharp drop-off in performance. This could be attributed to the fact that when there are 9 or more threads, a thread may be put to sleep while holding a contentious lock, prolonging the period of time during which other threads cannot acquire said lock.

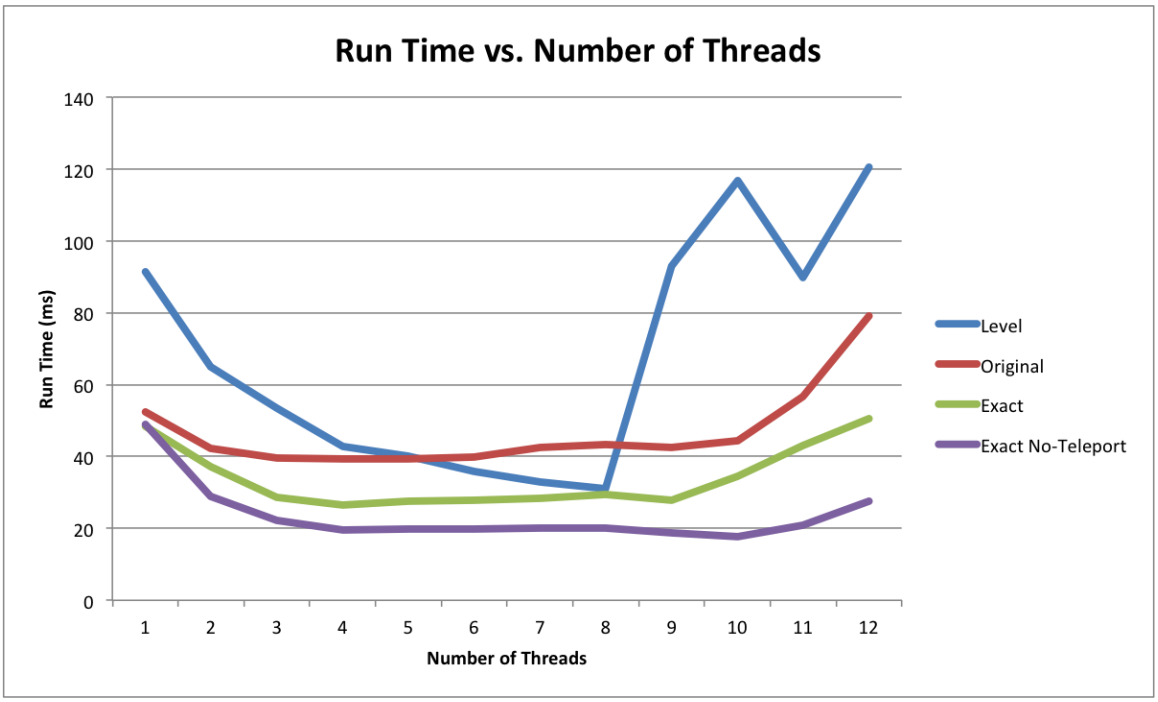

Figure 2-4: Run time versus number of threads.

The trends that we see in the run time graph are corroborated by those that appear in our graph of commit rate versus number of threads, Figure 2-5. First, we note the drastic decrease in commit rate, for all three skiplist variations, as the number of threads increases from 1 to 4 (or 1 to 8, in the case of the level skiplist). The fact that the commit rate appears to be inversely related to the number of threads indicates that many of our transaction aborts are due to conflicts between threads – oftentimes, we suspect, because of attempts to lock already-locked nodes within a transaction. (We discuss the commit rate issue in greater depth in the Results section.) Comparing the commit rates between the three variations, we see that the exact skiplist tends to have a slightly higher commit rate than does the original skiplist. The difference in their commit rates is quite small, but the exact skiplist's better commit rate helps to explain its outperformance of the original skiplist and makes sense given that fewer nodes are locked in the exact skiplist's `Find` algorithm, slightly decreasing the chances of transactions aborting. We also observe that the level skiplist has a shallower slope of commit rate degradation than do the other two variations. This indicates that the mutation algorithms employed in the level

skiplist are less sensitive to conflict aborts and might explain why we see greater relative performance in the level skiplist with higher thread counts (up to 8 threads), as compared to the exact and original skiplists.

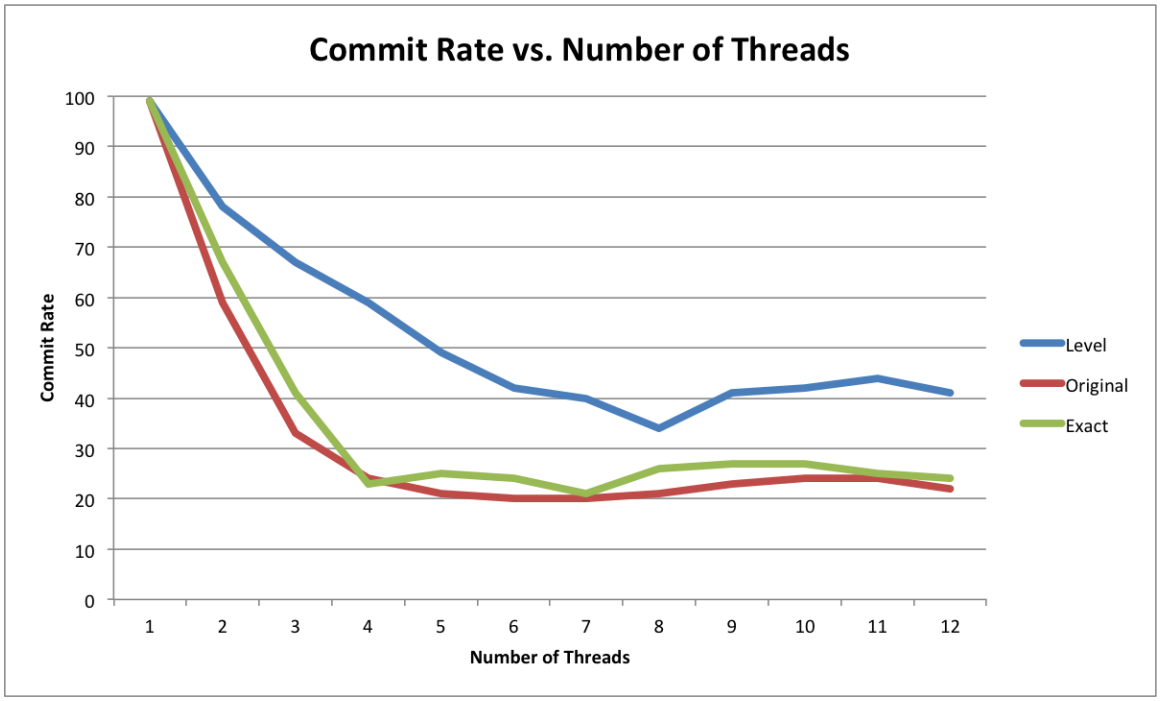

Figure 2-5: Commit rate versus number of threads.

Finally, we turn our attention to the graph of teleport length versus number of threads, Figure 2-6. We see a story that is similar to that which we observed in the commit rate graph. There is a marked decline in teleport length as the number of threads increases, but the decline is slightly less severe for the level skiplist than for the other two. Recall that our AIMD strategy for determining teleport length is based almost entirely off of commit rate – we halve our teleport length when a transaction aborts and increment it if a transaction commits (and has used up the entire allotted traversal limit). As a result, it is probable that the declining teleport lengths are largely a reflection on the commit rates. Thus, we draw the same conclusion here that we drew previously: all three skiplist variations suffer from a great deal of transaction aborts caused by conflicts between threads (namely, as stated earlier, attempts during transactions to lock already-locked nodes), but the level skiplist is less affected and thus we see that its performance benefits more from increases in thread count.

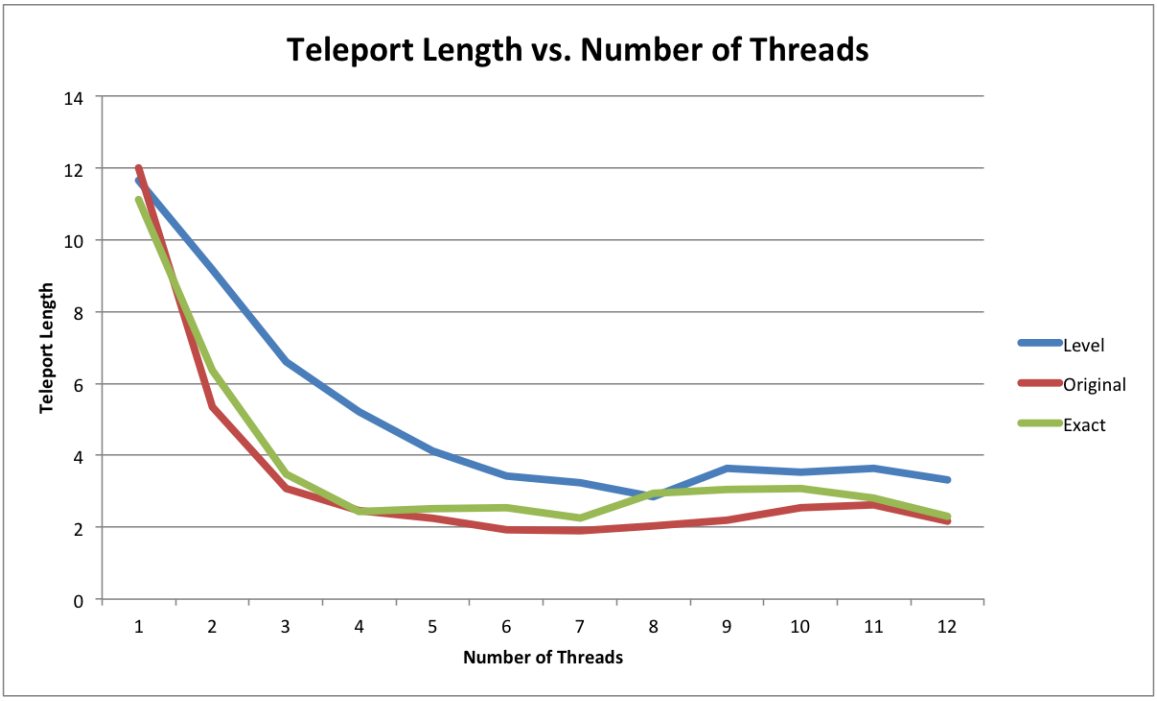

Figure 2-6: Teleport length versus number of threads.

2.5 Discussion

The results that we presented in the previous section do not look promising for the teleporting lock-coupled skiplist. One of the main issues that we run into with the three variations of the teleporting lock-coupled skiplist that we present above is a commit rate that plummets as we increase the number of threads accessing and mutating the skiplist concurrently. Recall, as we described in the beginning part of this chapter, that our `Teleport` method functions by searching down the skiplist without obtaining any locks until it either reaches the teleport limit or finds the node for which it is searching. At this point in our code, in order to maintain the invariant that all of the predecessors of the victim node are locked upon being found, we try to acquire locks for the predecessors that we encountered during the teleport. However, our transaction is guaranteed to abort if any of the nodes that we attempt to lock are either already locked when we begin our transaction or become locked while our transaction is taking place. This is doubly bad. When a transaction aborts, not only does the thread responsible for the transaction not hold locks for any of the predecessors that it found, it also loses its place further down the skiplist and must begin searching from the location at which it entered the transaction. Note that this

is distinctly different from what happens when a thread searches the skiplist non-transactionally. In a non-transactional traversal, the consequence of a node's lock being held by a different thread is simply that the current thread must wait until the lock is released, which means that in the non-transactionally based lock-coupled search, there is no possibility of losing work that has already been done.

Our graphs show us the negative effects of a having a significant amount of contention on nodes (particularly high-level nodes) within the skiplist and our reasoning about lock acquisitions in transactions gives us insights into why transaction aborts are costly (and frequent) in our teleporting skiplist code. With these understandings in mind, we propose adapting our `Teleport` method to be free of lock acquisitions. Transactions are not well-suited to performing lock acquisitions (especially when contention is high) because they cannot wait on a lock to be released, but must instead abort and retry (or revert to non-transactional code), significantly increasing the cost of encountering a locked node. In our teleporting exact skiplist code, we are able to reduce the number of locks that a thread must acquire during the course of a search, thus reducing the amount of contention in the skiplist. Likewise, in our teleporting level-locking skiplist code, we attempt to reduce contention by assigning one lock to each level of each node, making our code more fine-grained. While both strategies have the potential to alleviate contention to the point of outperforming the non-teleporting skiplist, we have not yet seen success in practice. (Although we should remind the reader that a similar lock-teleportation strategy used in [2] has been successful in improving the performance of the lock-coupled linked-list, as mentioned in the introductory chapter.)

While teleportation does not produce good performance in situations where contention is high and locks must be acquired within a transaction, it is still quite promising as a strategy for avoiding some of the intermediate work that is usually required to traverse a skiplist. In fact, in the next chapter, we present our hazard pointer teleportation algorithm, which proves to be effective in boosting the performance of the lazy skiplist.

Teleporting Lazy Skiplists

We now discuss our implementation, written in `C++`, of the teleporting lazy skiplist. We assume prior knowledge of the lazy skiplist (and refer the reader to [1, ch.14] for background information on this skiplist). In this chapter, we first cover the design of the teleporting lazy skiplist and its nodes before providing an overview of the general algorithm that we use for hazard pointer teleportation and a brief description of the skiplist's `Add`, `Remove`, and `Contains` methods. We then discuss variations on the teleporting lazy skiplist. In the Benchmarking, Results, and Discussion sections, we describe the performance measures that are of interest to us, provide graphs of our findings using these measures, and give an interpretation of these findings.

3.1 Design of Skiplist and Nodes

We rely heavily on the teleporting lock-coupled `Skiplist` in our design of the teleporting lazy `Skiplist`. The teleporting lazy `Skiplist` `Node` structures have several new members that serve to enable lazy locking, in addition to those present in the teleporting lock-coupled `Skiplist`. We also add a hazard pointer array, *hazard*, to the `Skiplist` structure to allow for threads to publish the addresses of the `Node`s that they need to hazard protect. Our *hazard* array is set up such that for each thread, for each level of the skiplist, we have a set of two hazard pointer slots. In our discussion of the algorithm, we explain the usage of the *hazard* array and why we require two slots for every thread and level combination.

3.2 Algorithm

We now turn our attention to the hazard pointer teleportation algorithm that we utilize. The design of the `Add`, `Remove`, and `Contains` methods is based on [1, ch.14]; we will give a brief description of the first two.

3.2.1 Find Method and Teleportation

As in the teleporting lock-coupled skiplist, the `Add`, `Remove`, and `Contains` methods of the teleporting lazy skiplist all depend on the `Find` method. The `Find` method in the teleporting lazy skiplist builds closely off of that in the lock-coupled skiplist (see Figure 2-1), but additionally takes in an array of node pointers, *succs*, and an `Int` pointer, *level*. It populates *succs* with successor nodes, which we define to be exactly those nodes pointed to by the predecessors' next pointers. (Thus, the victim node is always the lowest-level successor node.) In *level*, we store the height of the victim node, if the victim's value matches that for which we are searching. These additions require only trivial changes to the teleporting lock-coupled skiplist's version of `Find`.

The `Fallback` and `Teleport` methods of the teleporting lazy skiplist do not differ drastically from those in the teleporting lock-coupled skiplist either. However, in the teleporting lazy skiplist we must be certain that each node that we want to read is hazard protected prior to being read. Thus, each thread publishes, in an alternating manner, to the two *hazard* array slots that correspond to that thread and the skiplist level across which it is currently traversing. We require two slots for each thread and level combination because we need to ensure that all of the victim's predecessors and successors remain hazard protected from the time we encounter them until we are able to lock all of the victim's predecessors. This necessity informs the structure of the *hazard* array.

When dealing with hazard pointer memory management, we must also keep track of which nodes are currently hazard protected. Thus, outside of the `Fallback` and the `Teleport` methods, we maintain the invariant that all of the predecessors pointed to by the *preds* array and all of the successors pointed to by the *succs* array are hazard protected. This invariant holds when we enter and exit the `Fallback` and `Teleport` methods.

The `Fallback` method for the teleporting lazy skiplist functions in nearly the same manner as its lock-coupled counterpart's `Fallback` method (Figure 2-2). That is, it performs one individual step of the skiplist traversal, either following one next pointer or moving down one level of the skiplist. However, when it follows the node's next pointer, it calls `HazardRead` rather than performing a regular read. The `HazardRead` method loads the address of the node in question into the less-recently-used of the two slots that correspond to the current thread and level in *hazard*, overwriting the address of whichever node was previously hazard protected in that slot. It then executes a memory barrier. The only other difference in the lazy `Fallback` method

```
bool teleport(int v, Node* preds[], Node* succs[],
              ThreadLocal* threadLocal, int* start_layer, int* level) {
1    if (threadLocal->teleportLimit < 0) {
2      threadLocal->teleportLimit = DEFAULT_TELEPORT_DISTANCE;
3    }
4    int dist = threadLocal->teleportLimit;
5    int curr_layer;
6    Node* pred = preds[*start_layer];
7    for (int layer = *start_layer; layer >= 0 && dist > 0; layer--) {
8      curr_layer = layer;
9      Node* curr = pred->next[layer];
10     preds[layer] = pred;
11     while (curr->value < v && --dist > 0) {
12       preds[layer] = pred = curr;
13       curr = curr->next[layer];
14     }
15     succs[layer] = curr;
16     if (*level == -1 && curr->value == v) {
17       *level = layer;
18     }
19   }
20   bool done = curr_layer == 0 && preds[0]->next[0]->value >= v;
21   for (int i = *start_layer; i >= curr_layer; i--) {
22     transactionalHazardRead(&preds[i], threadLocal);
23     transactionalHazardRead(&succs[i], threadLocal);
24   }
25   *start_layer = curr_layer;
26   _xend();
27   if (dist  <= 0) { threadLocal->teleportLimit++; }
28   return done;
29 }
```

Figure 3-7: The lazy `Teleport` method, used in `Find`.

is that there is an extra conditional in the code to populate the *level* variable, if necessary. We omit the code for the lazy `Fallback` method because of its similarity to its lock-coupled counterpart.

The reader will notice the lazy `Teleport` method likewise shares many similarities with the lock-coupled `Teleport`. See Figure 3-7 for the code. We note that the main difference occurs in lines 21-24, after the *preds* and *succs* arrays have been populated during skiplist traversal. We do not need to hazard protect any of the nodes that we read within the transaction because the transaction operates on a frozen memory state. However, in order to maintain our invariant that all nodes pointed to by the *preds* and *succs* arrays are hazard protected after `Teleport` finishes, we must publish the addresses of the nodes that we added to these arrays within the transaction. We do not need a memory barrier following the publication of these addresses because, if the transaction commits, the changed state will become immediately accessible to the other threads. In `TransactionalHazardRead` we simply load the address of the specified node into the appropriate slot in *hazard*.

3.2.2 Add and Remove Methods

The `Add` and `Remove` methods for the teleporting lazy skiplist employ the same algorithm as that described in [1, ch.14], but have been refactored to some extent. We only describe them briefly here. The `Contains` method has not been changed; we do not describe it here.

The `Add` method (Figure 3-8) takes in an `Int`, v, and a `ThreadLocal` structure, *threadLocal*. We performs several initializations and then, in line 6, we enter into the main loop of the method. Within the loop, we find the victim node in line 7. In lines 9-16, we check whether the value, v, that we are looking for is in the skiplist, both physically and logically. If it is, then we return *false*. If it is in the process of being removed, then we loop through and call `Find` again. If it is not in the skiplist, then in line 18 we call `LockAndStrongValidate`. This method attempts to lock all of the predecessor nodes and validate that each still points to the successor node that we found during the `Find` method. It returns *true* holding locks for all of the predecessors on success, and *false* holding locks for none of the predecessors on failure. If `LockAndStrongValidate` is successful, then in lines 19-26 we allocate a new node, link it in to the skiplist, unlock the predecessors, and report our success. If `LockAndStrongValidate` is unsuccessful then either at least one node has been added in between the predecessor and successor nodes that we found, or at least one

of the predecessor or successor nodes has been removed. In both cases, we must start the loop from the beginning again.

The `Remove` method (Figure 3-9) shares several similarities with the `Add` method, the first being its method signature. The important part of the method begins on line 7, after initializations have occurred. We call `Find` to locate our victim. In lines 10-12, if it is our first run through the loop and the node with value v is either physically absent from the skiplist, logically absent from the skiplist, or has changed height since we found it (which indicates that it was added and removed in the meantime), then we report that it is not in the skiplist. In lines 13-21, if the node with value v is, indeed, in the skiplist during the first run through, then we lock it, ensure that it is still in the skiplist, and logically remove it. After this point, our goal is to physically remove the node from the skiplist. We attempt this by calling `LockAndWeakValidate` on line 22. The `LockAndWeakValidate` method is the same as the `LockAndStrongValidate` used in `Add`, except for the fact that we use weaker validation criteria for each of the predecessors. If `LockAndWeakValidate` succeeds, then our predecessor nodes are all locked and, in lines 23-29, we can unlink the node, retire it, and unlock both the node and its predecessors. If `LockAndWeakValidate` fails, then we must loop through again, this time only calling `Find` and then `LockAndWeakValidate`.

3.2.3 Variations

We have explored several variations on the teleporting lazy skiplist described above. We have experimented with limiting the number of hazard reads performed during the `Find` method, manually overwriting hazard pointers after they are no longer needed (rather than waiting for a new hazard pointer publication to prompt an overwrite), and with variations on the AIMD strategy (described in the previous chapter) for determining the teleport limit. In our experience, none of these alterations significantly affected the performance of the teleporting lazy skiplist, although further experimentation might yield better results. Furthermore, it is possible that performance improvements to our baseline lazy skiplist implementation could provide some benefit to the teleporting lazy skiplist.

3.3 Benchmarking

We use the same benchmarking suite that we use for the teleporting lock-coupled skiplist. Just as with the teleporting lock-coupled skiplist, we keep the benchmark

```
bool add(int v, ThreadLocal* threadLocal) {
1    Node* preds[MaxHeight];
2    Node* succs[MaxHeight];
3    Node* curr;
4    int height = randomLevel();
5    int level = -1;
6    while (true) {
7      find(v, preds, succs, threadLocal, &level);
8      curr = succs[0];
9      if (curr->value == v) {
10       if (curr->marked) {
11         PAUSE();
12         continue;
13       }
14       while (!curr->fullyLinked) {}
15       return false;
16     }
17     int top = MaxHeight;
18     if (lockAndStongValidate(preds, succs, 0, MaxHeight)) {
19       Node* node = allocate(v, height, threadLocal);
20       for (int layer = 0; layer < height; layer++) {
21         node->next[layer] = succs[layer];
22         preds[layer]->next[layer] = node;
23       }
24       node->fullyLinked = true;
25       unlock(preds, 0, top);
26       return true;
27     }
28     PAUSE();
29   }
30 }
```

Figure 3-8: The lazy `Add` method.

```
bool remove(int v, ThreadLocal* threadLocal) {
1    Node* preds[MaxHeight];
2    Node* succs[MaxHeight];
3    Node* curr;
4    bool firstTime = true;
5    int level= -1;
6    int start = 0;
7    while (true) {
8      find(v, preds, succs, threadLocal, &level);
9      curr = succs[0];
10     if (firstTime && (curr->value != v || !curr->fullyLinked
           || (curr->height)-1 != level || curr->marked)) {
11       return false;
12     }
13     if (firstTime) {
14       curr->lock();
15       if (curr->marked) {
16         curr->unlock();
17         return false;
18       }
19       firstTime = false;
20       curr->marked = true;
21     }
22     if (lockAndWeakValidate(preds, succs, 0, MaxHeight)) {
23       for (int layer = 0; layer < curr->height; layer++) {
24         preds[layer]->next[layer] = curr->next[layer];
25       }
26       retire(curr, threadLocal);
27       curr->unlock();
28       unlock(preds, 0, top);
29       return true;
30     }
31   }
32 }
```

Figure 3-9: The lazy `Remove` method.

parameters fixed unless we specify otherwise. (Recall: the skiplist initial size is 5,000 nodes; we run 100,000 operations for each benchmark test; our node values are between 1 and 10,000; and our skiplist height is 32.) Again, we present only the results for a read probability of 50%, as our results look very similar for both high and low read probabilities.

We benchmark the teleporting hazard-protected lazy skiplist ("hazard-teleport") against the regular hazard-protected lazy skiplist ("hazard"), which, as we will see, the hazard-teleport skiplist is able to outperform. For comparison we also include the following skiplists in some of our tests: the lazy skiplist with no hazard-protection ("simple lazy"); our implementation of the hazard-protected lazy skiplist that has been trivially converted to a lazy skiplist without hazard-protection ("adapted lazy"); the exact lock-coupled skiplist, which was our gold-standard in the previous chapter, ("exact no-teleport"); and the teleporting exact variation from the previous chapter, which was our most performant of the teleporting lock-coupled skiplists ("exact").

3.4 Results

We now turn our attention to the results of benchmarking the hazard-teleport skiplist against the other skiplist variations. To begin, we look at performance. In Figure 3-10, we graph the run time versus number of threads for several different skiplists. We first compare the hazard skiplist and the hazard-teleport skiplist. Our results are good, as the hazard-teleport skiplist outperforms its non-teleporting counterpart across all thread counts. This is precisely what we had hoped to see. We also note that the performance of the hazard-teleport skiplist improves with increases in the number of threads – not a particularly surprising result but one that tells us that the benefit of greater parallelization due to higher numbers of threads outweighs the increased contention that may occur. (Recall that the performance results for two of the teleporting lock-coupled skiplist variations did not display this characteristic.)

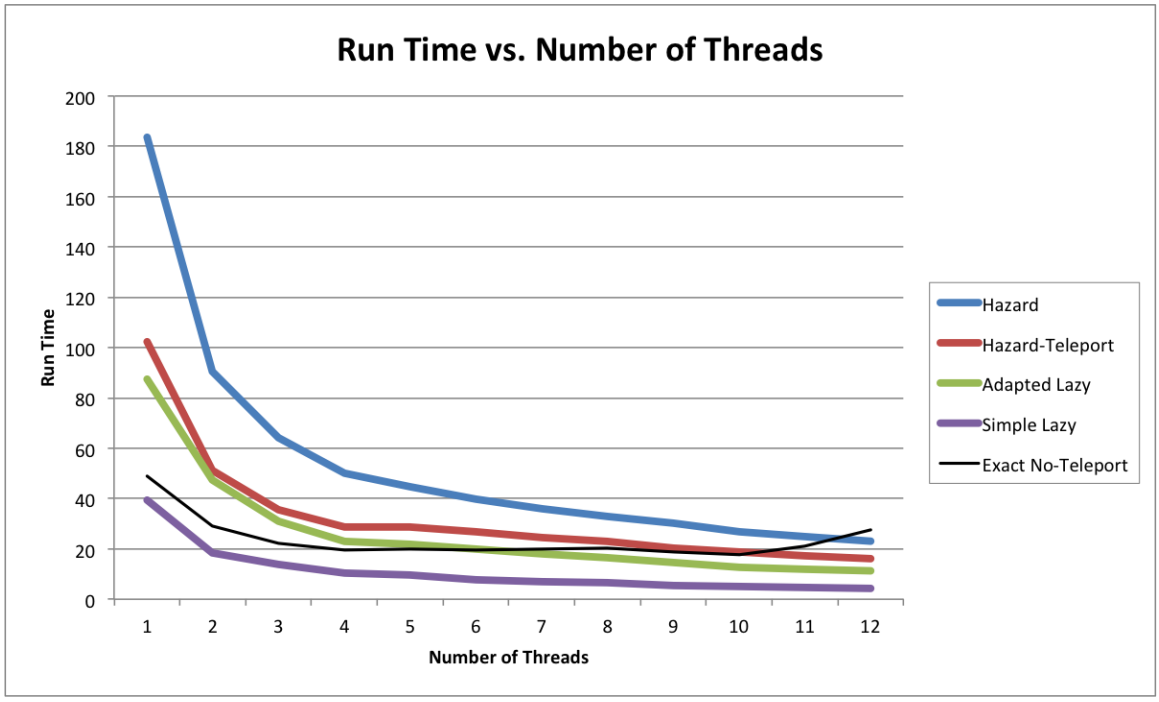

Figure 3-10: Run time versus number of threads.

Comparisons between the hazard-teleport skiplist and other skiplist variations plotted on our graph give us further information about the performance of the former. We note that the run times for the hazard-teleport skiplist follow those of the adapted-lazy skiplist quite closely across the various thread counts. The hazard-teleport skiplist uses exactly the same methods as does the adapted-lazy skiplist, except that it uses both hazard protection and teleportation. This tells us that the performance impacts of hazard protection (which imposes a performance cost) and teleportation (which improves performance) seem to nearly cancel each other out. When we look differences in run times for the adapted-lazy skiplist and the hazard skiplist, we can see just how much time is wasted performing intermediate hazard reads. Our teleportation method is able to eliminate nearly all of this wasted time.

Additionally, we consider the performance of the exact no-teleport skiplist – the skiplist which we were trying to outperform in the previous chapter. The comparison between this skiplist and our hazard-teleport skiplist is rather inexact, as the two skiplists use completely different algorithms, and one has memory management while the other does not. Nonetheless, it is exciting to see that the hazard-teleport skiplist, at thread counts of 11 and 12, is faster than the exact no-teleport skiplist at any thread count.

The final skiplist that we graph in comparison to the hazard-teleport skiplist is

the simple lazy skiplist, which has no teleportation and no hazard protection memory management. We include this skiplist because we find its comparison to the adapted lazy skiplist, which is also non-teleporting and without memory management, insightful. We hope that future iterations of the adapted lazy skiplist (which, unlike the simple lazy skiplist, can be trivially converted to a hazard-teleport skiplist) can be made to be closer in speed to the simple lazy skiplist. If this is possible, then it should follow that we also improve the performance of the hazard-teleport skiplist, possibly even to the point where its run time is as close to the simple lazy skiplist as it currently is to the adapted lazy skiplist.

The graphs of commit rate versus number of threads and teleport length versus number of threads support the good results that we saw in the run time graph. In both the commit rate and teleport length graphs, we plot the exact skiplist, which was the most performant teleporting lock-coupled skiplist discussed in the first chapter. In Figure 3-11, we can see that, while the hazard-teleport skiplist does suffer a small decline in commit rate as the number of threads increases, its decline is much less severe than that of the exact skiplist and its commit rate remains above 90% across all thread counts. Similarly, in Figure 3-12, we see that the average teleport distance of the hazard-teleport skiplist remains steady as the number of threads increases, a result that provides a stark contrast to the rapid fall in teleport length exhibited by the exact skiplist. The performance trends for the commit rate and teleport length of the hazard-teleport skiplist are as good as we can expect to see – it would be baffling if the commit rate or teleport length increased with increasing numbers of threads, as we expect higher thread counts to produce more contention.

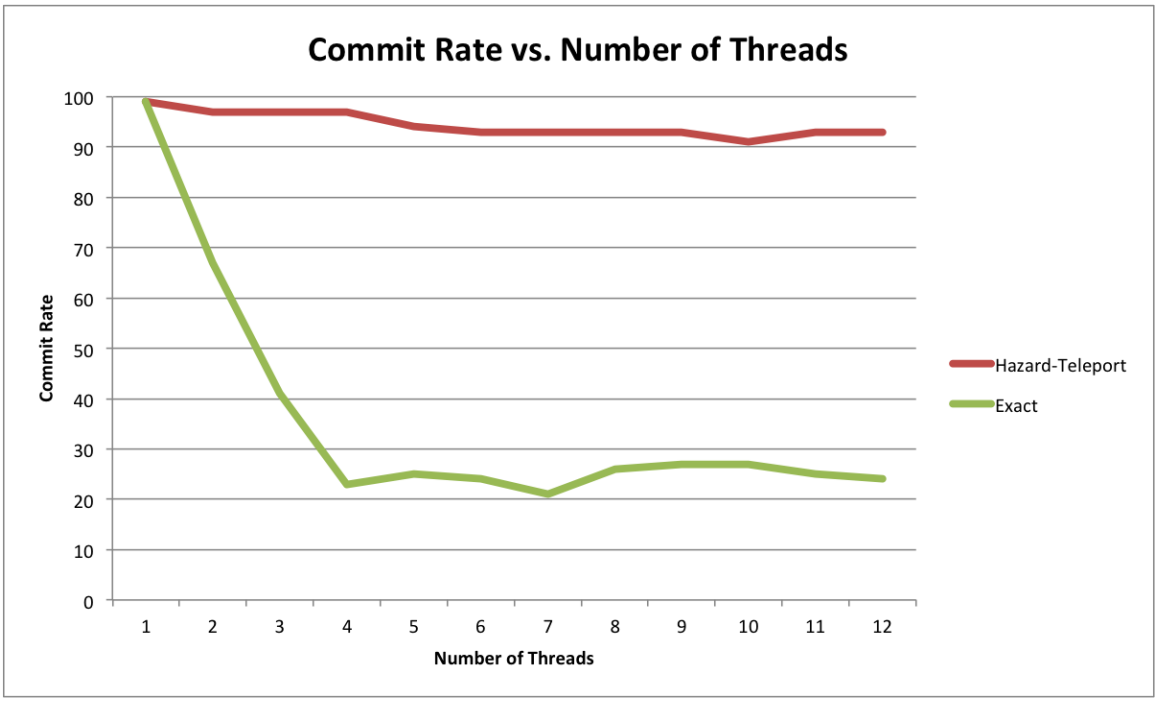

Figure 3-11: Commit rate versus number of threads.

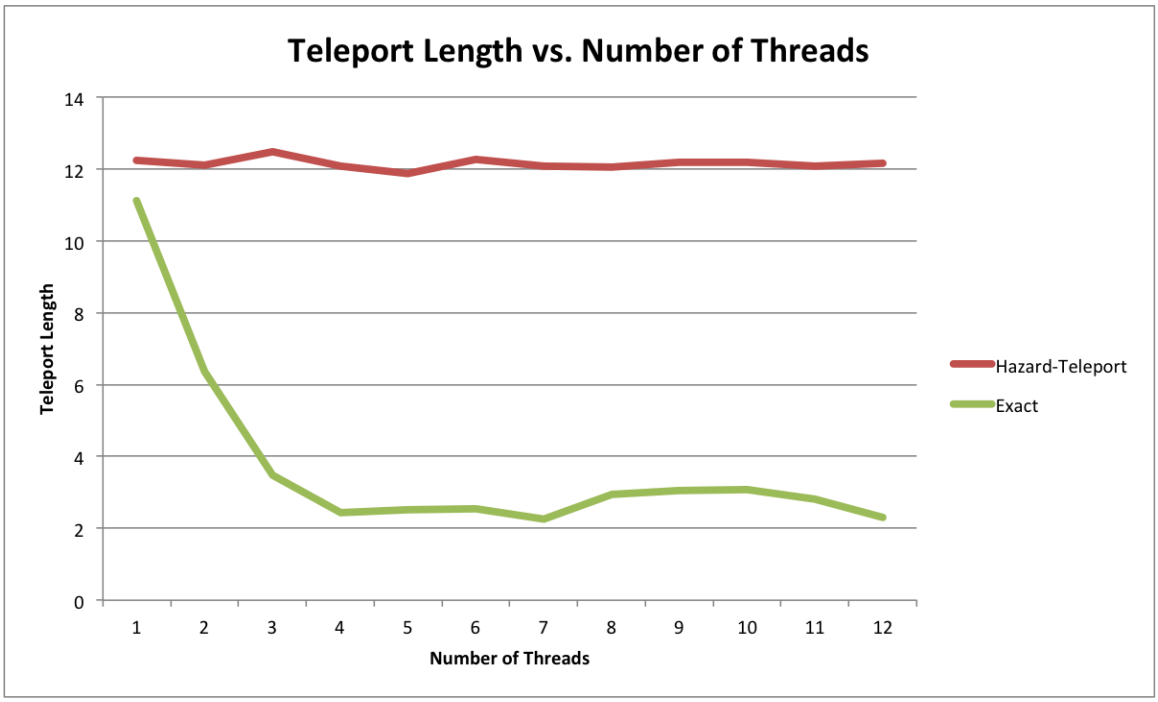

Figure 3-12: Teleport length versus number of threads.

3.5 Discussion

The Results section demonstrates that our hazard pointer teleportation method is indeed effective in improving the performance of the hazard-protected lazy skiplist. We also see from our results that the code parallelizes well and does not suffer from much contention. The teleporting hazard-protected lazy skiplist is almost as fast as the lazy skiplist that does not employ memory management – a result that is markedly different from that which we observed for the teleporting lock-coupled skiplist.

Hazard pointer teleportation is effective for the skiplist data-structure, whereas lock-teleportation is not, because of the differences between hazard pointer publication and lock acquisition. We discussed, in the previous chapter, why lock acquisition might contribute to high abort rates for transactions and a great deal of lost work in high contention environments. Hazard pointer publication, on the other hand, is well adapted for use inside of transactions. Unlike lock acquisition, hazard pointer publication is immediate and is not affected by the value currently residing in the hazard pointer slot (or any other value, for that matter). In addition, hazard pointer publication does not significantly increase the chances of a transaction aborting. A thread that publishes a hazard pointer *does* increase the size of its write-set for the transaction. However, its write-set will never conflict with the write-set of another thread (assuming no false sharing) because each thread only writes to its own slots in the *hazard* array. Furthermore, its write-set will only conflict with the read-set of another thread when that thread reads the *hazard* array to determine which nodes are free to be reused. This only occurs during `Add` operations, at most once per operation. It represents only a small amount of the work that is done by a thread during an `Add` operation, thus the chances that this read will occur while another thread is in a transaction are low, making aborts fairly rare, as we saw in Figure 3-11. These characteristics of hazard pointer publication explain the good performance that we achieved with teleporting hazard-protected lazy skiplist.

Conclusion

Our results from the previous two chapters provide us with useful information about teleportation. We know, from past work on teleportation, that its use produces performance gains in lock-coupled linked-lists, and both hazard protected lazy and lock-free linked-lists. From the work in this thesis, we learn that lock teleportation is not easily adapted from linked-lists to lock-coupled skiplists, because of the number of lock acquisitions on highly contentious nodes that are necessary within the skiplist lock teleportation transactions. This leads us to believe that lock teleportation may only be a viable strategy for performance improvements in data-structures with low contention and in which contention is spread out evenly across nodes, as opposed to being centered around specific nodes (i.e. the high-level nodes in the skiplist).

When we turn our attention to the teleporting lazy skiplist, we discover that the effectiveness of hazard pointer teleportation observed in lazy linked-lists does, indeed, extend to skiplists. Furthermore, based on our analysis, in the Discussion section of chapter 2, of the high commit rate that we observed in the teleporting lazy skiplist, we suspect that hazard pointer teleportation might also be applicable to other linked-node data-structures where the memory management strategy employs hazard-reads to traverse intermediate nodes. We further suspect that the hazard pointer teleportation algorithm that we developed for the lazy skiplist might be suitable, with minimal changes, to the hazard protected lock-free skiplist. This is an area of ongoing work.

Our research into improving skiplist performance through teleportation is carried out with the broader goal in mind of making the best use of current hardware to produce data-structures that can support the increasing efficiency demands in software. We hope that a better understanding of the viability of teleportation and a successful teleporting lazy skiplist implementation can contribute to this effort.

Bibliography

[1] Maurice Herlihy & Nir Shavit, *The Art of Multiprocessor Programming*, Burlington, MA, 2008

[2] Elias Wald & Maurice Herlihy, *State Teleportation via Hardware Transactional Memory*, 2015. Forthcoming.